VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Implementace komponenty tunelovacího serveru pro Virtlab

Implementation of Network Traffic Forwarding Component for Virtlab

2012 Daniel Stříbný

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Daniel Stříbný**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Implementace komponenty tunelovacího serveru pro Virtlab
Implementation of Network Traffic Forwarding Component for Virtlab

Zásady pro vypracování:

Cílem práce je reimplementace komponenty tunelovacího serveru pro Virtuální laboratoř počítačových sítí (Virtlab) s pomocí nástrojů, které jsou k dispozici v jádru operačního systému Linux. Nová implementace tunelovacího serveru, a to včetně jednotné konfigurační nádstavby, která bude z hlediska konfiguračního rozhraní zpětně kompatibilní s již existujícím řešením, by měla přinést především vyšší přepínací výkon a stabilitu oproti současnému stavu.

1. Seznamte se současným stavem řešení komponenty tunelovacího serveru. Proveďte analýzu požadavků na tuto komponentu a ověřte jejich splnitelnost za použití standardních nástrojů operačního systému GNU/Linux.
2. Navrhněte způsob realizace s ohledem na distribuovaný charakter systému. Při návrhu dbejte na snadnou rozšiřitelnost a modularitu celého řešení. Navržené řešení implementujte.
3. Proveďte důkladné testování vytvořeného díla a srovnejte dosažené výsledky s existujícím řešením.
4. Vytvořenou komponenty nasaďte do reálného provozu Virtlabu.

Seznam doporučené odborné literatury:

BENVENUTI, Christian. Understanding Linux network internals. USA : O Reilly Media, Inc., 2006. 1035 s. ISBN 9780596002558.
BECK, Michael. Linux kernel internals. 2nd. USA : Addison-Wesley, 1998. 480 s. ISBN 9780201331431.
MAMONE, Mark. Practical Mono. USA: Apress, 2005. 424 s. ISBN 9781590595480.
GUNNERSON, Eric. A Programmer's Introduction to C# 2.0, Third Edition. USA: Apress, 2005. 568 s. ISBN 9781590595015.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Martin Milata**

Datum zadání: 18.11.2011
Datum odevzdání: 04.05.2012

VYSOKÁ ŠKOLA BÁŇSKÁ-TECHNICKÁ UNIVERZITA * Fakulta elektrotechniky a informatiky * OSTRAVA *

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Souhlasím se zveřejněním této diplomové práce dle požadavků čl. 26, odst. 9 *Studijního a zkušebního řádu pro studium v magisterských programech VŠB-TU Ostrava.*

V Ostravě 4. května 2012 .....

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 4. května 2012 .....

Rád bych poděkoval především vedoucímu mé práce Ing. Martinu Milatovi za jeho neocenitelné rady a pomoc. Dále všem přátelům a kolegům za jejich podporu a rady.A v neposlední řadě také mé rodině, za jejich podporu v mém studiu.

Abstrakt

Cílem této práce je implementace komponenty tunelovacího serveru pro Virtlab. Obsahuje analýzu stávajícího řešení a návrh řešení nového. Dále výběr vhodného programovacího jazyka, nástrojů a technologií použitých k implementaci nového řešení. Také obsahuje popis nově implementovaného řešení.

Klíčová slova: tunelovací server, Virtlab, GRE, Mono, SNMP

Abstract

Goal of this diploma thesis is to implement network traffic forwarding component for the Virtlab laboratory. It contains analysis of the current solution and the design of the new solution. It also describes the selection of the suitable programming language, tools and technologies used for implementation of the new solution. It also contains a description of the newly implemented solution.

Keywords: tunnel server, Virtlab, GRE, Mono, SNMP

Seznam použitých zkratek a symbolů

.NET	–	Sada knihoven pro systém Windows
Mono	–	Multiplatfromní Open Source implementace .NET frameworku
Virtlab	–	Virtuální laboratoř počítačových sítí VŠB-TUO
C#	–	Objektově orientovaný programovací jazyk
GRE	–	Generic Routing Encapsulation - zapouzdřující protokol vyvinutý společností Cisco
SNMP	–	Simple Network Management Protocol - protokol pro vzdálený management
MTU	–	Maximum Transmission Unit - maximální velikost PDU
PDU	–	Protocol Data Unit - datová jednotka daného protokolu
VLAN	–	Virtual Local Area Network - virtuální lokální síť

Obsah

Seznam tabulek

Seznam obrázků

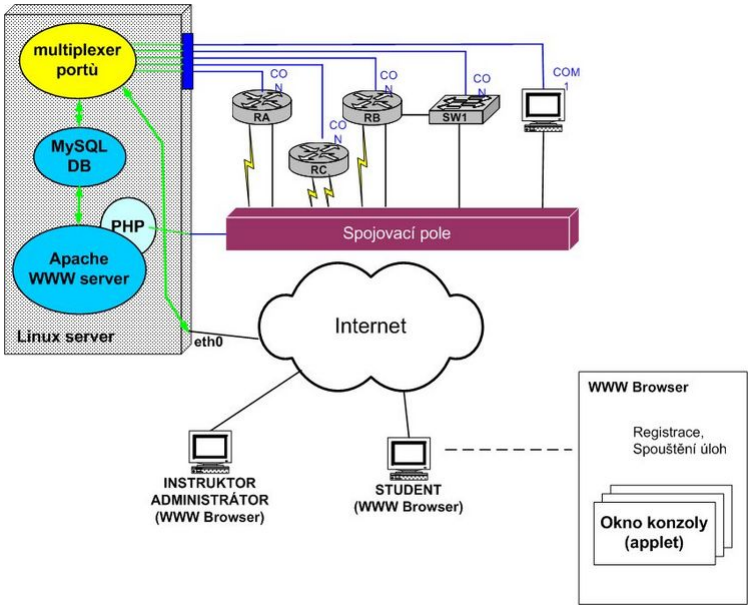
Seznam výpisů zdrojového kódu

1 Úvod

1.1 Virtlab

Projekt Virtlab, neboli virtuální laboratoř počítačových sítí, je zaměřen především na vzdálené zpřístupnění síťových prvků pomocí internetu. Vzniká na katedře informatiky VŠB-TUO od roku 2005 a slouží jako studijní pomůcka studentům, kteří tak mohou využívat pokročilá síťová zařízení bez nutnosti jejich zakoupení. V poslední verzi je Virtlab řešen jako distribuovaný projekt, jež združuje zařízení z několika vzdálených lokalit a umožňuje tak efektivnější využití síťových prvků. V systému jsou také zastoupeny virtualizované síťove prvky a stanice.

Virtlab vzniká především prostřednictvím diplomových a bakalářských prací a je tedy rozdělen na jednotlivé spolupracující části, které nezbytně nemusí sdílet programovací jazyk [5].



Obrázek 1: Architektura Virtlabu [5]

1.2 Tunelovací server

Jednou z mnoha klíčových komponent Virtlabu je tunelovací server. Jeho hlavní funkcí je spojování prvků, které jsou umístěny v různých vzdálených instancích Virtlabu.

Konkrétně jde o propojování ethernetových portů daných prvků na 2. vrstvě ISO-OSI modelu. Tunelovací server tedy vytváří jakési pseudodráty, simulující přímé propojení prvků kabelem. Jeho cílem by také mělo být, aby uživatel nebyl schopen při zevrubném průzkumu tento propoj rozpoznat.

1.3 Cíle práce

Vzhledem k ne příliš vydařenému návrhu původního tunelovacího serveru je cílem této práce upravit jeho návrh a provést reimplementaci.

Hlavní cíle práce jsou následující:

- Analýza současného stavu
- Provedení nového návrhu komponenty
- Zvolení vhodných nástrojů a programovacího jazyka
- Implementace

Dalšími cíli jsou:

- Využít stávající hardwarovou a softwarovou platformu tunelovacích serverů
- Implementovat řešené s ohledem na modularitu a jednoduchost
- Zachovat zpětnou kompatibilu s ostatními částmi Virtlabu
- Využít pouze Open Source technologie
- Využít existujících (vyzrálých) nástrojů
- Při volbě programovacího jazyka zohlednit budoucí možnost reimplementace celého projektu

1.4 Přehled práce

První kapitola této práce obsahuje motivaci a cíle, které si autor klade. Dále také zevrubný popis Virtlabu a význam tunelovacího serveru.

Druhá kapitola se zabývá analýzou prostředků a technologií pro řešení implementace komponenty a popisuje důvody pro jejich konkrétní výběr. Dále rozebírá požadavky, kladené na výsledné řešení.

Třetí kapitola se věnuje návrhu nového řešení. Také popisuje teoretické řešení jednotlivých případů užití a výber programovacího jazyka.

Čtvrtá kapitola obsahuje implementaci navrhovaných řešení ze druhé kapitoly a je doplněna ukázkami zdrojového kódu.

Pátá kapitola se zabývá testováním výsledného řešení a popisuje také nutná nastavení pro nasazení.

Šestá kapitola popisuje nasazení výsledného řešení a možnosti budoucího rozvoje výsledného řešení

Sedmá kapitola obsahuje závěr a zhodnocení celé práce.

2 Analýza

2.1 Požadavky

Požadavky na tunelovací server souvisí úzce s jeho hlavní funkcí, tedy s vytvářením virtuálních propojů mezi instancemi Virtlabu, resp. zařízeními náležejícími k těmto instancím.

Mezi nejdůležitější požadavky patří:

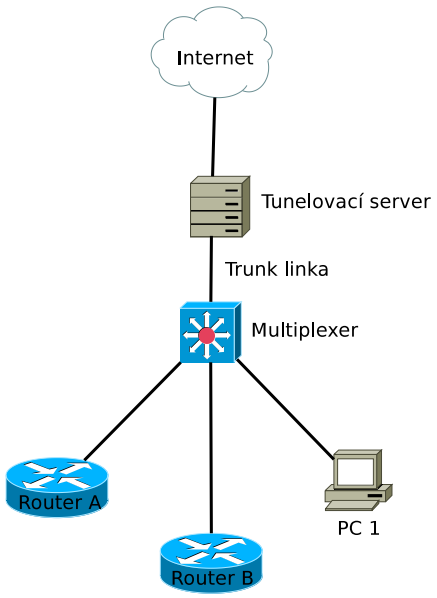
- Dobré parametry vytvořených virtuálních propojů - zejména zpoždění a využitelné MTU
- Stabilita propojů

Na další parametry, jako je například rychlost vytváření propojů, již není třeba klást přehnaný důraz.

2.2 Funkce tunelovacího serveru

Tunelovací server jako takový je se sestavá ze serveru s OS Linux na němž běží samotná aplikace a který je propojen skrze internet s ostatními lokalitami Virtlabu, konkrétně s jejich tunelovacími servery.

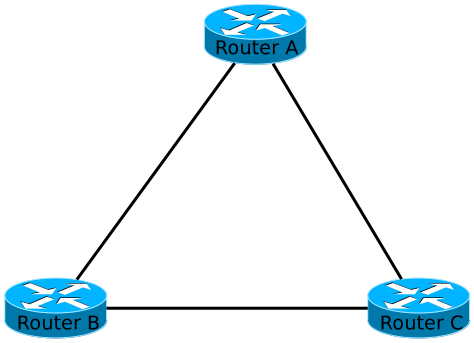
K tomuto serveru je připojen multiplexer portů, sestávající se z Multilayer přepínače, nakonfigurovaného k agregaci linek k jednotlivým síťovym zařízením do jedné trunk linky, která nese dílčí VLANy jednotlivých zařízení k ethernetovému rozhraní samotného serveru.



Obrázek 2: Základní koncept

Konkrétní příklad fuknce tunelovacího serveru ilustruji na jednoduchém příkladu.

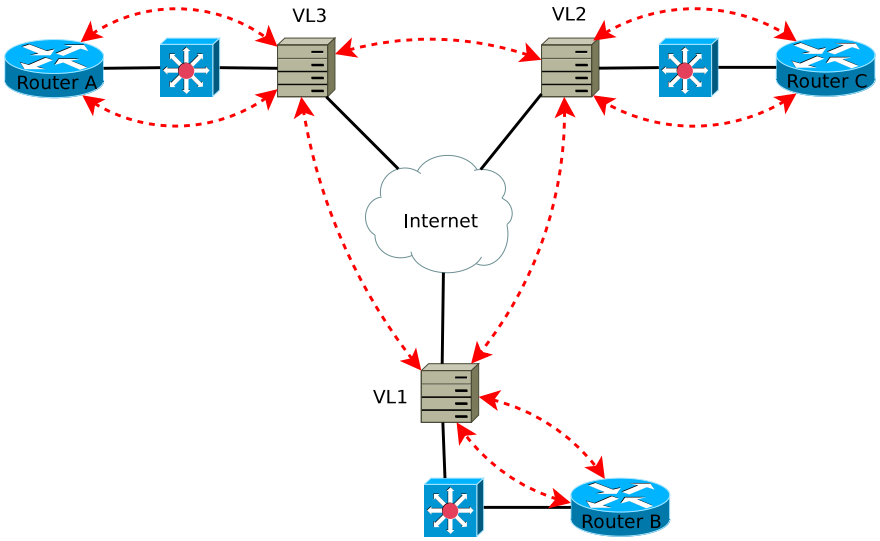
Uživatel chce vytvořit jednoduchou topologii ve tvaru trojúhelníku (obrázek 3), kdy jsou prvky (v tomto případě routery) propojeny ethernetovými kabely.



Obrázek 3: Ukázková topologie

Virtlab zajistil pro tuto topologii prvky ze tří oddělených lokalit. Tunelovací servery v jednotlivých lokalitách tedy musí zajistit vytvoření virtuálních propojů, které budou simulovat přímé propojení prvků.

Na obrázku 4 je ukázáno jak bude po vytvoření propojů tato topologie vypadat z pohledu Virtlabu.

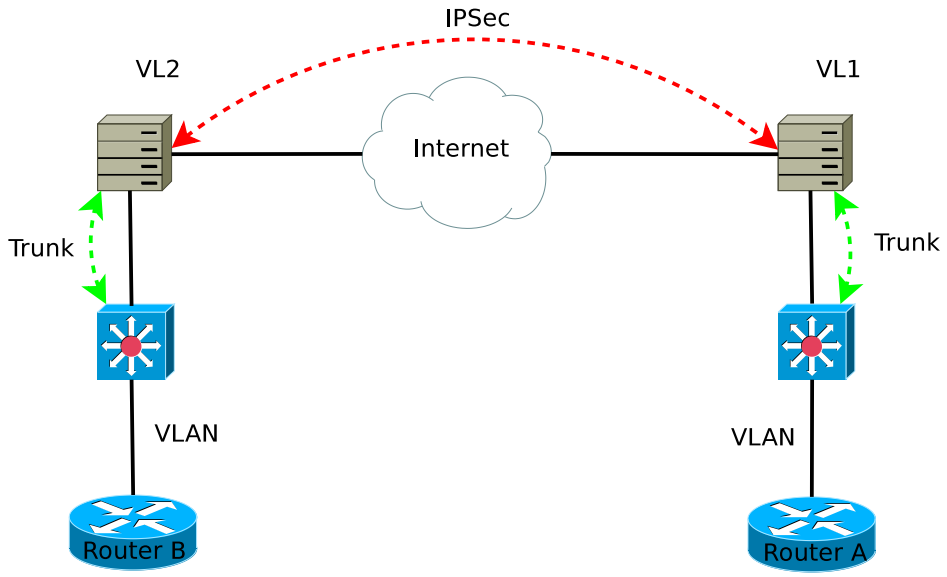


Obrázek 4: Ukázková topologie z pohledu Virtlabu

2.3 Původní řešení

Funkce původního řešení byla založena na výše zmíněném základním konceptu. K vytváření propojů mezi lokalitami se využívalo IPSec tunelu, jímž byly navzájem propojeny všechny

lokality. Při požadavku na vytvoření propoje byly L2 rámce z propojovaného prvku zachytávány na rozhraní s připojeným multiplexerem. Tyto pak byly encapsulovány do UDP packetů a odeslány skrze tunel k cílové lokalitě. V případě požadavku na propojení zařízení v rámci jedné lokality se předávaly rámce mezi jednotlivými VLANami v rámci trunk linky multiplexeru.



Obrázek 5: Původní řešené tunelovacího serveru

Nevýhoda spočívala především v nutnosti manuálního zpracovávání jednotlivých rámců samotnou aplikací a jejich následného odesílání ve fromě UDP packetů. Výkon byl tedy závislý na konkrétní implementaci. Také bylo nutno nastavit každou instanci Vitlabu samostatně, resp. samotné tunelovací servery neměly možnost mezi sebou vyjednat propoj. Při lokálním propojení se také zbytečně předávaly rámce až v samotné aplikaci tunelovacího serveru.

3 Návrh nového řešení

Nové řešení tunelovacího serveru by mělo odstraňovat nedostatky původního řešení se zachováním zpětné kompatibility se zbytkem systému.

Kvůli uvedeným nedostatkům, jsem se rozhodl při analyzování nového řešení připustit možnost nahrazení stávajícího konceptu encapsulace L2 rámců do UDP packetů při vytváření propojů vzdálených lokalit. S ohledem na snahu využít existující komponenty implementované pro OS Linux, jsem zaměřil pozornost na tyto protokoly tunelování L2 rámců:

- GRE
- L2TPv3

Postavením nového řešení na těchto existujících komponentech by se také odstranila účast samotné aplikace na přeposílání L2 rámců, což by mělo kladný vliv na její výkon.

3.0.1 GRE

GRE je protokol vyvinutý společností Cisco, sloužící k zapouzdření řady protokolů 2. a 3. vrstvy IOS-OSI modelu uvnitř virtuální Point-to-Point linky nad IP protokolem. Přes takto vytvořený propoj lze tedy tunelovat zejména rozšířené protokoly Síťové vrstvy IPv4 a IPv6, případně i protokol Ethernet pracující na Spojové vrstvě. Typické použití je při vytváření VPN na zmíněných vrstvách. Standard pro tento protokol je definován v RFC 2784 a následně rozšířen v RFC 2890 [7]. Podpora pro protokol GRE je integrována v jádru většiny distribucí operačního systému Linux prostřednictvím modulu *ip_gre*.

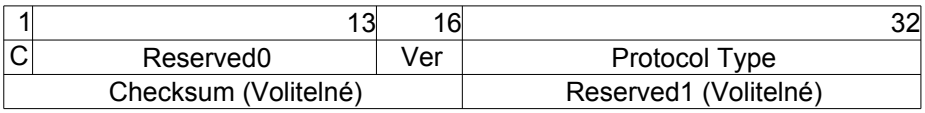
3.0.1.1 Funkce Protokol vytváří další vrstvu v klasickém dělení podle ISO-OSI modelu. Tedy vytváří jakýsi separátor a rozděluje toto dělení dále na 2 nezávislé protokoly, kdy jeden je nesen druhým. Zde záleží, zda je nesený protokol Spojové či Linkové vrstvy.

Vrstvy ISO-OSI modelu	
5	Relační
4	Transportní
3	Zapouzdřená síťová
2	Zapouzdřená linková
	Zapouzdření – GRE
3	Síťová
2	Linková
1	Fyzická

Obrázek 6: ISO-OSI model s vloženým GRE zapouzdřením přenášejícím 2. a 3. vrstvu

Význam jednotlivých částí hlavičky protokolu GRE zobrazené na obrázku 7:

- C - 1 bitová značka, označující přítomnost kontrolního součtu (Checksum)



Obrázek 7: Hlavička protokolu GRE

- Reserved0 - bity rezervovány pro různé další značky, jejichž použití musí implementovat jak vysílající tak přijímající zařízení
- Ver - verze protokolu, v současném standardu musí být nulová
- Protocol Type - označuje typ protokolu, kterému patří nesený packet (rámec)
- Checksum - kontrolní součet hlavičky GRE protokolu a neseného packetu (rámce)
- Reserved1 - rezervován pro budoucí použití, v současné verzi musí být nulový

Protokol je původně koncipován jako plně bezestavový. Tedy koncová zařízení tunelu nemohou zjistit případný výpadek zařízení na druhém konci tunelu. V posledních implementacích protokolu byla však přidána možnost odesílání keepalive zpráv. Tyto odesílají zapouzdřený GRE IP packet v dalším GRE zapouzdření (tedy dvojité zapouzdření). Přijímající strana pouze odstraní vnější zapouzdření a odešle odpověď přímo z fyzického rohraní, čímž je vyloučeno ovlinění odpovědi samtnou funkcí GRE protokolu. Další vlastností těchto keepalive zpráv je nezávislost jejich zapnutí na obou koncích tunelu. Je tedy možné zapnout keepalive pouze na jednom konci tunelu (využití v případě topolofií typu hub-and-spoke)[7]. Maximální využitelné MTU je u protokolu GRE 1462 Bytů.

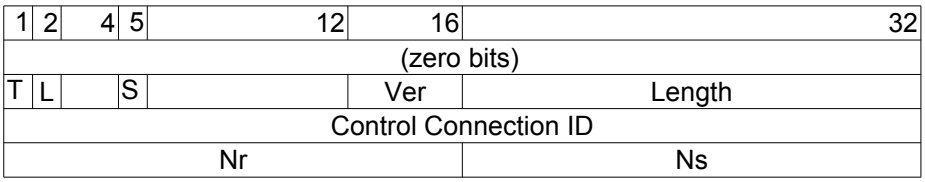
3.0.2 L2TPv3

Protokol L2TPv3 je, jak už název napovídá, 3. verzí tunelovacího protokolu L2TP. Stejně jako GRE poskytuje virtuální propoje, avšak s omezením pouze na protokoly 2. vrstvy ISO-OSI modelu. Je navržen jako odlehčená alternativa protokolu MPLS. L2TPv3 je nesen protokoly UDP nebo IP, pracujícím na 4. resp. 3. vrstvě ISO-OSI modelu. Standard pro tento protokol je definován v RFC 3931 [7]. Podpora pro protokol L2TPv3 je integrována v jádru operačního systému Linux prostřednictvím modulů *l2tp_eth* a *l2tp_ip*. Bohužel však většina distribucí neobsahuje jádro s podporou pro tyto moduly a je tedy nutná jeho rekompilace ze zdrojových kódů.

3.0.2.1 Funkce Protokol opět vytváří další vrstvu v klasickém dělení podle ISO-OSI modelu. Při své funkci však využívá kromě standartních zpráv, nesoucích samotná data zapouzdřeného protokolu, také zprávy sloužící k řízení své funkce.

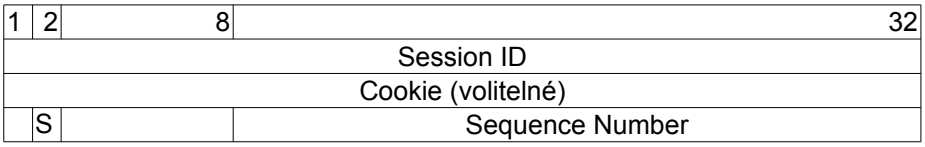
Význam jednotlivých částí hlavičky řídící zprávy protokolu L2tpv3 z obrázku 8:

- 32 nulových bitů - Session ID označující řídící zprávu (pouze pro L2tpv3 přes IP)
- T - bit označující řídící zprávu



Obrázek 8: Hlavička protokolu L2tpv3 - řídící zpráva

- L - bit značící přítomnost pole s délkou zprávy
- Ver - verze protokolu, pro L2tpv3 vždy nastavena na hodnotu *3*
- Control Connection ID - identifikace konkrétního kontrolního spojení
- Nr - sekvenční číslo zprávy
- Ns - sekvenční číslo očekáváné v následující zprávě



Obrázek 9: Hlavička protokolu L2tpv3 - datová zpráva

Význam jednotlivých částí hlavičky datová zprávy protokolu L2tpv3 z obrázku 9:

- Session ID - číslo relace, k níž náleží zpráva
- Cookie - přiřazuje zprávu k dané relaci - doplňující identifikátor k session ID
- S - bit označuje přítomnost sekvenčného čísla
- Sequence Number - sekvenční číslo, určující pořadí zpráv

Protokol nabízí především spolehlivý přenos dat, díky využití kontrolního součtu v případě využití protokolu UDP, v případě využití IP je tato vlastnost do určité míry suplována řídícími zprávami. Cenou za to je vyšší zatížení linek samotných protokolem, díky využití zmíněných řídících zpráv. Funkce protokolu je oproti GRE značně pokročilejší a její popis není cílem této práce. Maximální využitelné MTU při použití tohoto protokolu je, stejně jako v případě GRE, 1462 Bytů, v případě použití IP jako transportního protokolu [7].

3.0.3 Porovnání tunelovacích protokolů

Oba výše uvedené protkoly nabízí vytváření tunelů, nesoucích protokoly 2. vrstvy ISO-OSI modelu. U L2TPv3 se navíc jedná o spolehlivější a propracovanější vituální spoje. Maximální MTU je u obou protokolů shodné.

Pro využití v tunelovacím serveru Virtlabu byla klíčová podpora pro tyto protokoly v jádře operačního systému Linux. V případě nasazení L2TPv3 by bylo nutné rekompilovat jádra stávajích tunelovacích serverů, což nebylo žádoucí. Proto byl pro realizaci zvolen prtokol GRE. V případě budoucí potřeby, použít L2TPv3 tunelování, je jeho nasazení do existující aplikace poměrně jednoduchou záležitostí.

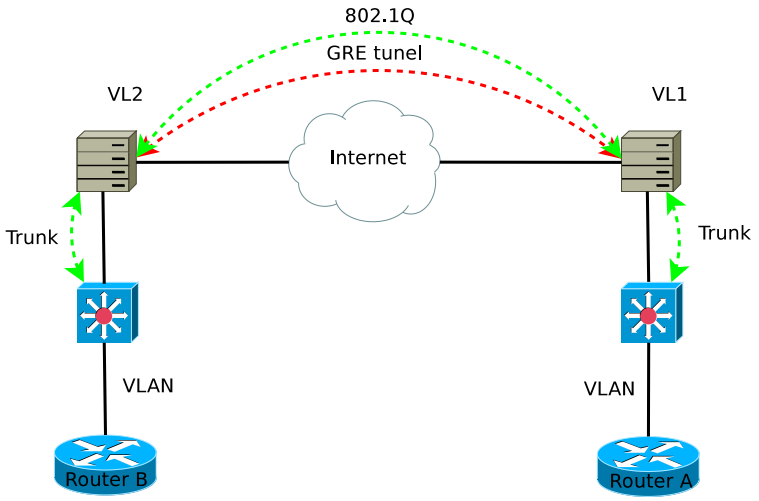
3.0.4 Vytváření vícenásobných propojů

Problémem, plynoucím z nutnosti zachovat stávající hardwarovou a softwarovou platformu, byla nutnost využít 1 veřejné IP adresy na každém existujícím tunelovacím serveru. Díky tomuto omezení bylo možno mezi jednotlivými tunelovacími servery vytvořit pouze jeden tunel. Jako optimální řešení bylo tedy možno zvolit jeden z těchto scénářů:

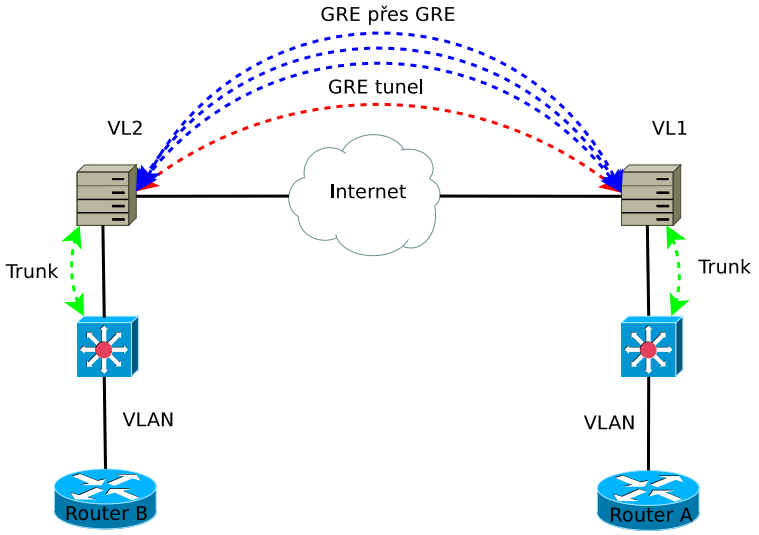
- Využít možnosti dvojího zapouzdření a tedy v hlavním GRE tunelu, vytvořeném mezi lokalitami s pomocí jejich veřejných IP adres, tvořit další GRE tunely s využitím adres privatního rozsahu přidělenému každému tunelovacímu serveru 11
- Využít rozdělení provozu v hlavním GRE tunelu značkováním L2 rámců pomocí standardu IEEE 802.1Q (tzv. VLAN tagging) 10

Výhodou VLAN značkování je větší využitelné MTU (1462 Bytů oproti 1438 Bytům u dvojího GRE zapouzdření). Naopak nevýhodou tohoto značkování je omezený počet vytvářených VLAN. U dvojího zapouzdření je také nutné nastavit směrování tunelovacího serveru s ohledem na dostupnost privátních rozsahů IP adres použitých v ostatních lokalitách.

S ohledem na tyto skutečnosti jsem zvolil VLAN značkování. Pro použití v současném stavu Virtlabu, je plně dostačující maximální možný počet VLAN sítí. V případě nedostatku je možné malou úpravou přejít na dvojí zapouzdření GRE případní na kombinaci obou řešení.



Obrázek 10: Návrh vytváření propojů pomocí technologie 802.1Q



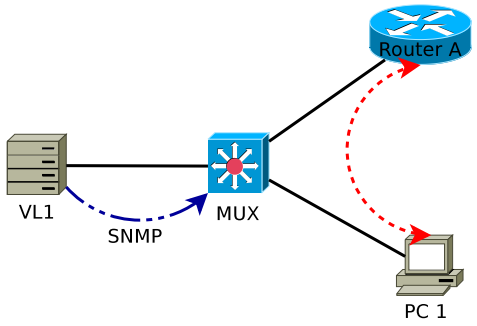
Obrázek 11: Návrh vytváření propojů s využitím vícenásobného GRE zapouzdření

3.1 Scénáře vytváření virtuálních propojů

S ohledem na funkci původního řešení tunelovacího serveru a požadavek na její zefektivnění jsem dospěl k těmto scénářům vytváření virtuálních propojů:

3.1.1 Propoj zařízení v rámci lokální instance Virtlabu

Propoj zařízení v rámci lokální instance Virtlabu je nutno řešit ve dvou samostatných případech. V nejjednodušším případě je potřeba propojit dvě koncová zařízení, bez nutnosti zachytávat provoz na vytvořením propoji. Zde lze vycházet z faktu, že každé zařízení v současném konceptu připojené k instanci Virtlabu má definovánu svou VLAN. V případě existence více Multiplexerů v rámci jedné instance jsou tyto navzájem propojeny Trunk linkou, která nese rámce všech VLAN k těmto multiplexerům připojených. Lze tedy jednuduše provést propojení dvou prvků změnou VLAN ID, na Multiplexerech připojených zařízení, na shodná.

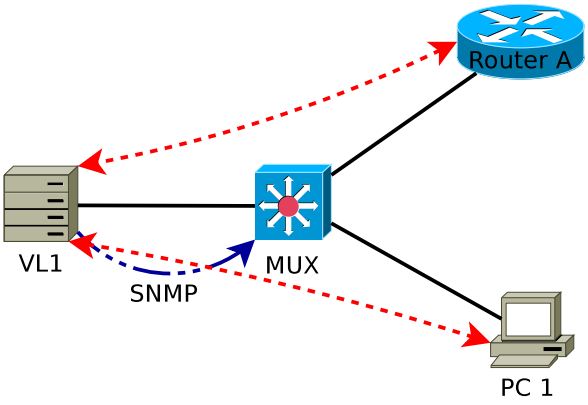


Obrázek 12: Propojení lokálních zařízení

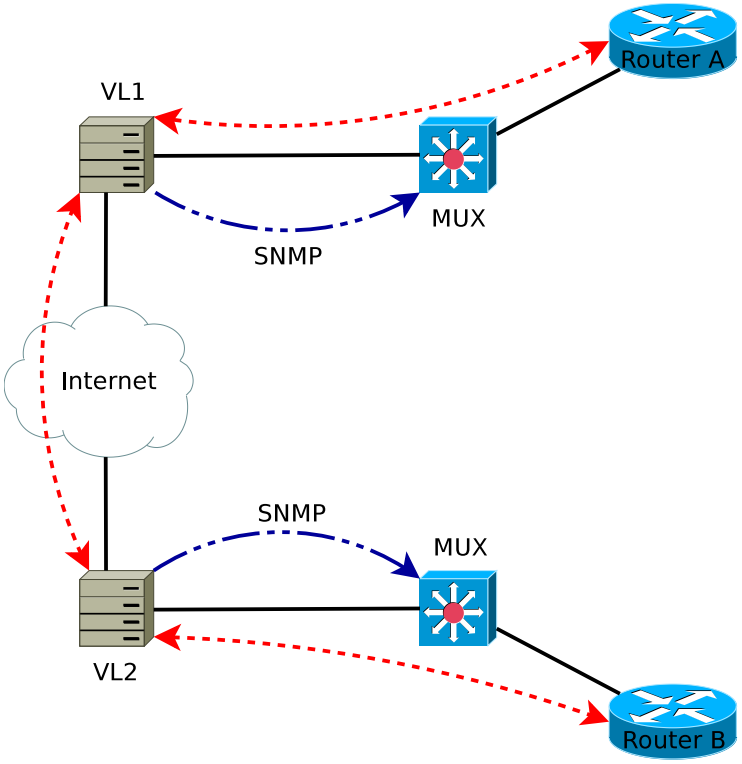
V případě nutnosti zachytávání provozu na daném provozu se propojení VLAN daných zařízení provede na samotném tunelovacím serveru. V tomto případě lze vyjít z konceptu původní implementace, kdy jsou rámce VLAN všech zařízení, dané instance Virtlabu, neseny Trunk linkou až na fyzické rozhraní tunelovacího serveru. Na tomto rozhraní se vytvoří rozhraní pro každou z propojovaných VLAN. Tyto jsou následně přemostěny rozhraním typu Bridge, na kterém jsou následně zachytávány rámce.

3.1.2 Propoj zařízení v odléhlé instanci se zařízením v lokální instanci Virtlabu

Pokud jsou propojovaná zařízení umístěná v různých instancích , je nutné vytvořit propoj také mezi těmito lokalita. Zde se aplikuje, výše zmíněná metoda, vytváření VLAN na GRE tunelu, propojujícím jednotlivé instance Virtlabu. Po vytvoření patřičných VLAN rozhraní na obou koncích tunelu, je nutné vytvořit rovněž rozhraní pro VLAN propojovaných zařízení na rozhraních s připojenou Trunk linkou k Multiplexeru. Následně jsou vytvořeny přemosťující rozhraní, pro propojení konců tunelů a odpovídajících VLAN rozhraní pro propojované prvky.



Obrázek 13: Propojení lokálních zařízení s možností zachytávání provozu

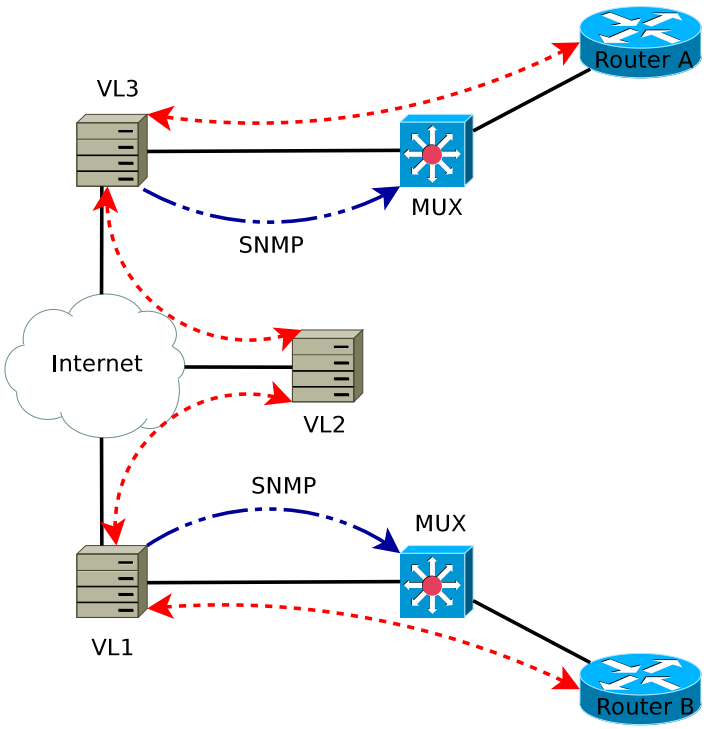


Obrázek 14: Propojení lokálního a vzdáleného zařízení

V případě potřeby zachytávání provozu, je provoz zachytáván na přemosťujícím rozhraní lokální instance Virtlabu.

3.1.3 Propoj zařízení ve dvou odlehlých instancích

V případě instance Virtlabu, která nemá žádná vlastní lokální zařízení, je postup obdobný s předchozím případem. Na vzdálených instancích, ve kterých se nacházejí propojované prvky, jsou vytvořeny odpovídající VLAN rozhraní na Trunk linkách k Multiplexerům, tyto jsou poté přemostěny s VLAN rozhraními na koncových rozhraních propojujích GRE tunelů. Na lokální instanci Virtlabu jsou následně přemostěny koncová rozhraní těchto GRE tunelů.



Obrázek 15: Propojení dvou vzdálených zařízení

V případě potřeby zachytávání provozu, je provoz opět zachytáván na přemosťujícím rozhraní lokální instance Virtlabu. V případě tohoto scénáře je možný alternativní postup, kdy je propoj veden přímo mezi vzdálenými instancemi. Není však poté možno zachytávat provoz na lokální instanci.

3.2 Výběr programovacího jazyka, nástrojů a technologií

Výběr programovacího jazyka u každého projektu ovlivnují především požadavky na podobu a vlastnosti výsledného produktu. Neméně důležitým kritériem jsou ovšem také znalosti programátora.

Vzhledem k mé oblibě programovacího jazyku C#, potažmo tedy i platformy .NET, jsem od počátku zvažoval implementaci v tomto jazyku. Vzhledem k nutnosti použití

Open Source řešení a nasazení produktu na platformě Linux jsem nakonec k implementaci zvolil projekt Mono.

Projekt Mono si klade za cíl vytvoření sady vývojových nástrojů a běhového prostředí, bitově kompatibilního s platformou .NET. Je vyvíjen od roku 2004 a na jeho vývoji se podílela společnost Novell společně s početnou komunitou nezávislých vývojářů. V současné době je projekt veden společností Xamarin, založenou původním autorem projektu Miguel de Icaza. Mono je multiplatformní, s distribucemi dostupnými pro platformy Linux, Windows, MAC OS a další. Podporuje také řadu programovacích jazyků, počínaje C# a Java přes Python až po Ruby či F#. Jeho cílem je v současné době především plná implementace vlastností .NET Frameworku 4.0. Součástí tohoto cíle je také implementace API kompatibilního s API původního .NET Frameworku společnosti Microsoft [6].

Po analýze projektu jsem musel určit způsoby jakými budou realizovány výše uvedené scénáře s ohledem na možnosti, které jazyk C# v tomto projektu nabízí.

3.2.1 Požadavky

Použitý jazyk, potažmo tedy C# v projektu Mono, musí být schopen:

- Vytvářet GRE tunely
- Vytvářet rozhraní pro jednotlivé VLAN značkované sítě
- Přemosťovat jednotlivá rozhraní
- Nastavovat konfiguraci Multiplexerů

Vytváření a přemosťování jednotlivých rozhraní provádí přímo jádro systému Linux. Pro komunikaci s jádrem využívají programy v uživatelském prostoru především tzv. Netlink socketů. Tyto sockety jsou speciální verzi obecných socketů (používá AF_NETLINK socket family)[2].

Tento typ socketů bohužel není v projektu Mono přímo podporován. Jelikož knihovny zodpovídající za sockety jsou momentálně ve stavu, kdy jejich funkcionalita odpovídá jejich ekvivalentům ve frameworku .NET, je tento fakt pochopitelný (systém Windows využívá odlišného principu komunikace).Řešení tohoto problému se nabídlo v podobě využití externích nástrojů, které budou popsány dále.

Dalším požadavkem bylo nastavování konfigurace Multilayer přepínače sloužícího jako Multiplexer. Pro tento případ lze nejlépe využít protokolu SNMP, který byl k tomutu účelu přímo vytvořen. Podpora tohoto protokolu není přítomna přímo v projektu Mono, avšak jsou k dispozici Open Source knihovny pro tento protokol [6].

3.2.2 Použité nástroje a technologie

Vzhledem k určitým omezením, daným použitým programovacím jazykem, bylo nutné najít nástroje, které umožní dosažení očekávané funkcionality. Při hledání jsem se zaměřil především na nástroje intergrované v nejnovějších verzích distribucí operačního systému Linux.

3.2.2.1 Iproute2 Tento nástroj, integrovaný ve většině distribucí systému Linux, poskytuje možnost nastavovat a vytvářet rozhraní, přemosťovat je a mnoho dalších funkcí. Toho dociluje posíláním požadavků jádru systému přes výše zmíněné Netlink sockety. Nástroj v současné době vyvíjí Stephen Hemminger. Hlavním cílem nástroje je nahradit dosud hojně používané nástroje ifconfig, route, vconfig a další [10].

Iproute2 je robustní a spolehlivý konzolový nástroj. Nevýhodou je mnohdy špatná dokumentace, což je ovšem případ většiny, bouřlivě se rozvíjejících, Open Source projektů.

Ze schopností nástroje uvedu několik případů, které jsem využil při realizování výše uvedených scénářů vytváření virtuálních propojů

- Vytvoření VLAN rozhraní

  ```
  #ip link add link eth0 type vlan name eth0.20 id 20
  ```

  Tento příklad vytvoří VLAN rozhraní s *id 20*, které se bude jmenovat *eth0.20* a bude vytvořeno nad rozhraním *eth0*.

- Odstranění VLAN rozhraní

  ```
  #ip link delete eth0.20 type vlan
  ```

  Tento příklad odstraní existující VLAN rozhraní *eth0.20*.

- Vytvoření GRE tunelu

  ```
  #ip link add gre1 type gretap local 10.0.0.1 remote 192.0.2.1 ttl 255
  ```

  Pomocí tohoto příkazu bude vytvořen koncový bod tunelu *gre1*. Tunel bude definován lokální IP adresou *10.0.0.1* a vzdálenou IP adresou *192.0.2.1*. Ttl v zapouzdřujících IP packetech tunelu bude nastaveno na hdonotu *255*.

- Osdtranění GRE tunelu

  ```
  #ip link del gre1 type gretap
  ```

  Obdobně, jako v případě VLAN rozhraní, provede odstranění rozhraní GRE tunelu *gre1*.

- Nastavení rozhraní do stavu *up*

  ```
  #ip link set eth0 up
  ```

  Nastaví rozhraní *erh0* do stavu *up*.

- Nastavení MTU daného rozhraní

  ```
  #ip link set eth0 mtu 1500
  ```

  Nastaví MTU na rozhraní *eth0* na hodnotu *1500*.

3.2.2.2 Brctl Nástroj Brctl slouží především k vytváření logických rozhraní přemosťujících jiná rozhraní. Umožňuje také úpravu těchto rozhraní [10].

Tento nástroj sice není v běžných distribucích systému Linux dostupný ve výchozím stavu, avšak je jej možno získat instalací z repozitářů v rámci balíku *bridge-utils* pomocí příkazu

```
#apt-get install bridge-utils
```

Ze schopností nástroje opět uvedu několik případů, které jsem využil při realizování výše uvedených scénářů vytváření virtuálních propojů. Nutno dodat, že nástroj je zaměřen výhradně na mostová rozhraní a tedy jeho schopnosti jsou omezeny pouze na tato rozhraní na rozdíl od multifunkčního *iproute2*.

- Vytvoření a odstranění rozhraní typu Bridge

  ```
  #brctl addbr br1
  ```

  Vytvoří rozhraní *br1* typu bridge.

  ```
  #brctl delbr br1
  ```

  Analogicky toto rozhraní odstraní.

- Přídání a odstranění rozhraní určených k přemostění

  ```
  #brctl addif br1 eth0
  ```

  Přidá rozhraní *eth0* k přemosťujícímu rozhraní *br1*.

  ```
  #brctl delif br1 eth0
  ```

  Odstraní toto rozhraní z mostního rozhraní *br1*.

3.2.2.3 Modprobe Nástroj *modprobe*, je kozolová aplikace, jejímž určením je připojování a odpojování modulů k jádru operačního systému Linux. Autorem je Rusty Russell a v současné době je udržován v rámci balíku *module-init-tools* Jonem Mastersem[10].

Využití nástroje v aplikaci je omezeno na odstranění VLAN, GRE a mostních rozhraní, při spuštění programu.

Příklady použití nástroje

- Připojení a odpojení modulu

  ```
  #modprobe ip_gre
  ```

  Připojí modul *ip_gre* k jádru systému.

  ```
  #modprobe -r ip_gre
  ```

  Odpojení daného modulu.

3.2.2.4 SNMP Pro úpravu konfiguraci Multiplexerů jsem vybral protokol SNMP, konkrétně jeho nejnovější podobu SNMPv3. Prokol umožňuje sledovat síťové prvky a měnit jejich konfiguraci. Je definován v několika různých RFC. SNMPv3 je těmito RFC definováno jako tzv. Full Standard, což popisuje RFC s nejvyšším stupněm vyzrálosti.

Architektura protokolu je založena na dvojici Manager - Agent. Agent je software na spravovaném zařízení, který zařizuje získávání dat z tohoto zařízení. Manager je software pro příjem, zpracování a interpretaci těchto dat. Data jsou na spravovaném zařízení uložena v tzv. MIB, což je stromová databáze jednotlivých OID, tedy proměnných, které může číst a upravovat Agent. SNMP pracuje na 7. vrstvě ISO-OSI modelu, tedy na vrstvě Aplikační. Na Transportní vrstvě je přenášeno protokolem UDP a využívá porty 161 a 162[7].

Pro SNMP je definováno několik typů zpráv. Specifickou vlastností SNMPv3 je přidání autentizace a zabezpečením. Konkrétně podporuje tyto úrovně

- noAuthNoPriv - poskytuje autentizaci přes uživatelské jméno
- authNoPriv - poskytuje autentizaci pomocí MD5 nebo SHA algoritmů
- authPriv - poskytuje autentizaci pomocí MD5 nebo SHA algoritmů a kryptování pomocí algoritmu DES

Konkrétní použití protokolu SNMP v aplikaci tunelovacího serveru bude popsáno v kapitole 4.

4 Implementace

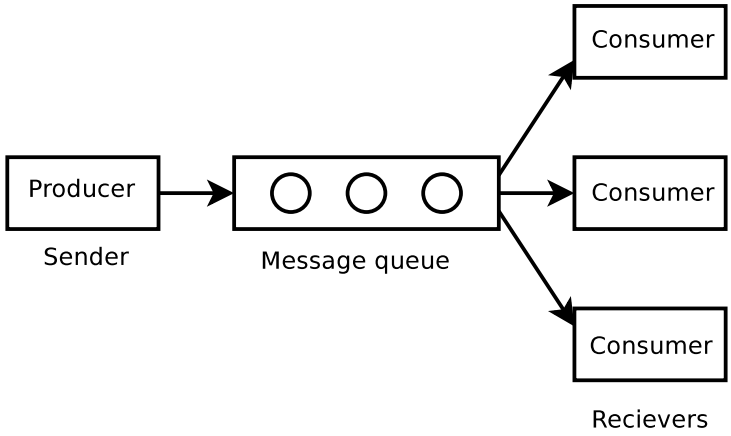
4.1 Vývojové prostředí

Při implementaci bylo jako vývojové prostředí použito IDE Monodevelop s frameworkem Mono ve verzi 2.10.8, běžící na operačním systému Ubuntu 11.10. Použitý jazyk byl již zmíněný C#. Projekt byl kompilován pro nejnovější verzi .NET frameworku 4.0.

Použité IDE je vyvíjeno v rámci projektu Mono. Poskytuje však mnohdy pouze základní funkcionalitu, oproti IDE Visual Studio společnosti Microsoft, které je určeno pro platformu Windows.

4.2 Architektura

Z výchozích požadavků a požadavků vyplývajících z analýzy, se jevilo zřejmé, implementovat aplikaci jako vícevláknovou. Jako výchozí návrhový vzor pro architekturu jsem použil vzor Producent-Konzument.



Obrázek 16: Návrhový vzor Producent - Konzument

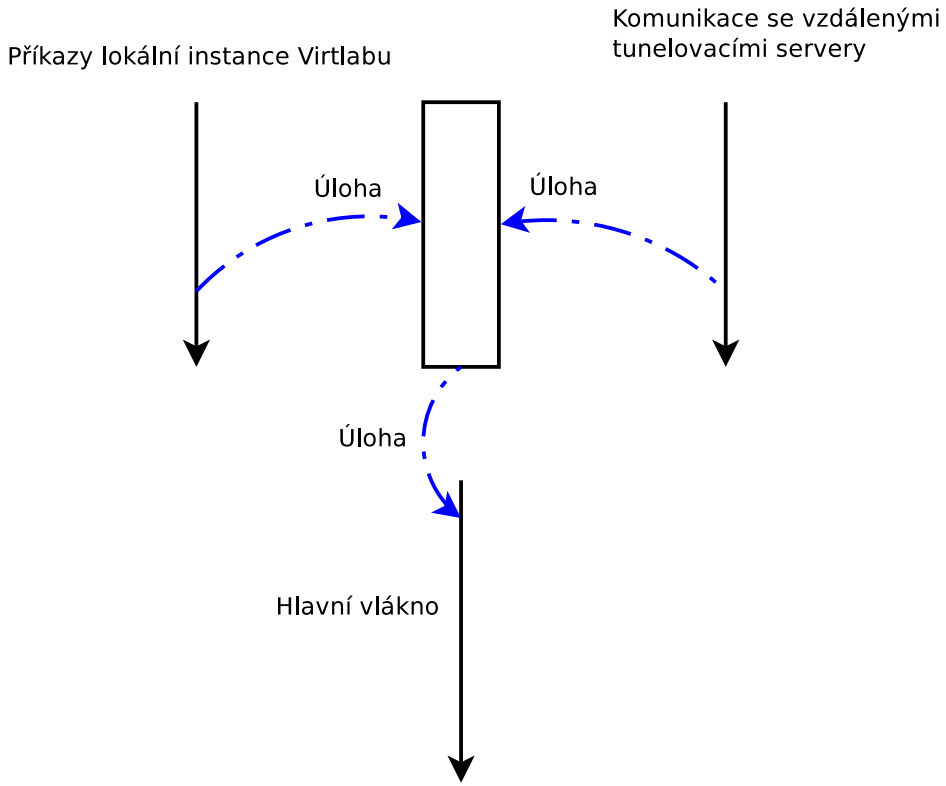
Program tedy je řešen 3 vlákny, kdy jedno vlákno je v roli konzumenta a zbylé jsou producenti. Vzhledem k malému běžnému výkonu producentů je toto řešení dostačující. V případě potřeby je možné, po malých úpravách, přidat více konzumujících vláken pro větší rychlost tunelovacího serveru.

Na obrázku lze vidět 2 produkující vlákna, které se starají o komunikaci s nadřízenými komponentami Virtlabu (pomocí protokolu TCP) a tunelovacími servery ve vzdálených lokalitách (komunikace pomocí protokolu UDP).

4.3 Struktura programu

Program se skládá ze 2 projektů. Tyto se dále dělí podle následující struktury.

- *CoreLogic*



Obrázek 17: Implementovaný návrhový vzor Producent - Konzument

  - *Comm.cs*
  - *Control.cs*
  - *CustomConfig.cs*
  - *Main.cs*

- *TunnUtil*
  - *DatabaseUtil.cs*
  - *ExternalToolsCalls.cs*
  - *LogUtil.cs*
  - *ParseUtil.cs*
  - *PcapUtil.cs*
  - *SnmpUtil.cs*
  - *Structs.cs*

Tímto rozdělením je zaručena jistá modularita programu. Program samotný je koncipován s určitou mírou jednoduchosti, tedy přílišná modularita je s tímto v přímém rozporu.

4.4 Jádro programu

Jádro programu je v architektuře kunzumujícím vláknem. Je tvořeno projektem typu konzolová aplikace a jsou v něm, kromě hlavní třídy Main, obsaženy také třídy sloužící ke komunikaci aplikace s nadřízenými komponentami Virtlabu a ostatními tunelovacími servery.

Je tvořeno projektem *CoreLogic*.

- *Comm.cs*
- *Control.cs*
- *CustomConfig.cs*
- *Main.cs*

4.4.1 Třída Main

Základem hlavní třídy *Main*, jádra aplikace, je nekonečná smyčka 1, která provadí vyzvedávání úloh z fronty úloh a jejich následné vykonávání. Smyčka zároveň přerušuje běh vlákna v případě, kdy není ve frontě k dispozici úloha ke zpracování. Úlohy jsou do fronty ukládány z vláken určených pro komunikaci s okolím, která budou popsána dále. Fronta je typu *ConcurrentQueue*, tedy neblokující, vláknově bezpečná fronta. Tento typ fronty byl přidán v *.NET 4.0*.

```
while (true) {
        if (mainQueue.TryDequeue (out tmp)) {

            Execute (tmp);

        } else {

            sig.Reset ();
            sig.WaitOne ();

        }
    }
```

Výpis 1: Hlavní smyčka

Tato hlavní třída také obsahuje většinu struktur, sloužících k ukládání stavu rozhraní tunelovacího serveru. Tyto struktury jsou popsány dále v této kapitole.

Dále třída *Main* obsahuje tyto metody

- *public static void Main (string[] args)* - Hlavní metoda aplikace, volaná při spuštění. Obsahuje především hlvní smyčku.
- *private static void LoadConfiguration()* - Provádí načtení konfigurace z konfiguračního souboru do připravených polí a proměnných.

- *private static void Execute(Task task)* - Zpracovává úlohu vyzvednutou z fronty a volá metody, které provádí proces vytvoření spojení.
  - Task task - úloha, vyzvednutá z frotny hlavní smyčkou programu
- *private static void CreateConnection()* - Zajišťuje vytvoření patřičných záznámů ve strukturách držících informace o vytvořených propojích. Tato metoda je přetížena.
  - *private static void CreateConnection(string[] input, Socket socket)* - Spouští iniciaci procesu vytváření propoje, v případě požadavku ze strany lokální instance Virtlabu/uživatele. Zajišťuje také oveření zda daný propoj již existuje a odeslání odpovědi uživateli.
    - string[] input - rozdělený vstupní řetězec
    - Socket socket - Socket, přes který je připojen uživatel/nadřazená komponenta Virtlabu
  - *private static void CreateConnection(Msg message, EndPoint ep)* Spouští iniciaci procesu vytváření propoje, v případě požadavku ze strany vzdálené instance Virtlabu.
    - Msg message - zpráva odeslaná vzdáleným tunelovacím serverem
    - EndPoint ep - lokace vzdáleného tunelovacího serveru
- *private static void ConnectionCreated(Msg message)* - Prováděna po potvrzení, zaslaném vzdálnou instancí Virtlabu. Dokončuje proces vytvoření spojení.
  - Msg message - zpráva, odeslaná vzdáleným tunelovacím serverem, potvrzující vytvoření spojení
- *private static void RemoveConnection()*- Zajišťuje zneplatnění patřičných záznámů ve strukturách držících informace o vytvořených propojích. Tato metoda je přetížena.
  - *private static void RemoveConnection(string[] input, Socket socket)* - Spouští iniciaci procesu odstranění propoje, v případě požadavku ze strany lokální instance Virtlabu/uživatele. O úspěšném odstranění informuje uživatele.
    - string[] input - rozdělený vstupní řetězec
    - Socket socket - Socket, přes který je připojen uživatel/nadřazená komponenta Virtlabu
  - *private static void RemoveConnection(Msg message, EndPoint ep)* Spouští iniciaci procesu vytváření propoje, v případě požadavku ze strany vzdálené instance Virtlabu.
    - Msg message - zpráva odeslaná vzdáleným tunelovacím serverem
    - EndPoint ep - lokace vzdáleného tunelovacího serveru
- *private static void ConnectionRemoved(Msg message)* - Dokončuje proces odstranění propoje

  - Msg message - zpráva, odeslaná vzdáleným tunelovacím serverem, potvrzující odstranění spojení

- *private static void ShowConnectionTable(Socket socket)* - Vypisuje uživateli obsah tabulky spojeni.
  - Socket socket - Socket, přes který je připojen uživatel.
- *private static void ShowHelp(Socket socket)* - Vypisuje uživateli nápovědu aplikace.
  - Socket socket - Socket, přes který je připojen uživatel.
- *private static string GenerateID()* - Pomocná metoda, vytvářející identifikátor propoje.
- *private static int Validate(String input)* - Validuje uživatelský vstup a vrací druh zadaného příkazu. Validace je řešena pouze základním způsobem, vzhledem k faktu, že s aplikací bude komunikovat pouze obsluha Virtlabu, nikoliv běžný uživatel.
- *private static void Setup()* - Nastavuje úvodní konfiguraci aplikace. Volá metodu LoadConfiguration(), pro načtení konfigurace. Dále vytváří objekty pro komunikaci s okolím a pro volání externích nástrojů. Navazuje také GRE tunely, pro každou vzdálenou instanci Virtlabu, načtenou z konfiguračního souboru.

Část metody pro vytváření propoje je možno nalézt ve výpisu 7.

Po spuštění programu je jako první volána metoda *Main*. V této je nejprve zavolána metoda *Setup*, které nastaví vše potřebné. Dále program pokračuje průchodem hlavní smyčky. Pokud čeká nějaká úloha ve frontě, je tato vyzvednuta a voláním metody *Execute* zpracována. Při zpracování je vyhodnoceno o jaký typ úlohy se jedná a následně je zavolána odpovídající metoda. Následně se provádí některý ze scénářů vytváření propoje, případně jeho odstranění.

Vzhledem k nutnosti, zachování kompatibilituy se stávajícím řešením, bylo nutné vyřešit problém vícenásobného vytváření stejných propojů. V původním řešení totiž nadřazené komponenty Virtlabu zasílaly požadavek na vytvoření propoje jak na lokální tunelovací server tak na tunelovací server vzdálené instance Virtlabu, která měla participovat na vytvářeném propoji. V nové implementaci tunelovacího serveru však jednotlivé tunelovací servery vyjednávají propoje nezávisle na nadřazených komponentách, tedy by při inicializaci vprocesu vytváření propoje na obou praticipujících instancích Virtlabu docházelo k vytvoření dvou duplikítních propojů. Toto je vyřešeno porovnáním časů inicializace procesu vytvářeného propoje, kdy později vytvářený propoj není dokončen.

Validace uživatelského vstupu je řešeno poměrně stroze, vzhledem k výlučnému použití uživatelského vstupu administrátorem Virtlabu a nadřazenými komponentami Virtlabu. Základní validace je řešena pomocí regulárních výrazů. Dále je ošetřeno navazování více stejných propojů. Další validace se vzhledem k odbornosti obsluhy, případně přesnosti nadřazených komponent Virtlabu, jeví zbytečná.

4.4.2 Třída CustomConfig

V třídě *CustomConfig* je definována struktura konfiguračního souboru aplikace. Instance této třídy je využita v metodě *LoadConfiguration* v rámci třídy *Main*.

V samotné třídě jsou definovány konfigurační elementy, kolekce těchto elementů a sekce konfiguračního souboru, ve kterých se tyto vyskytují.

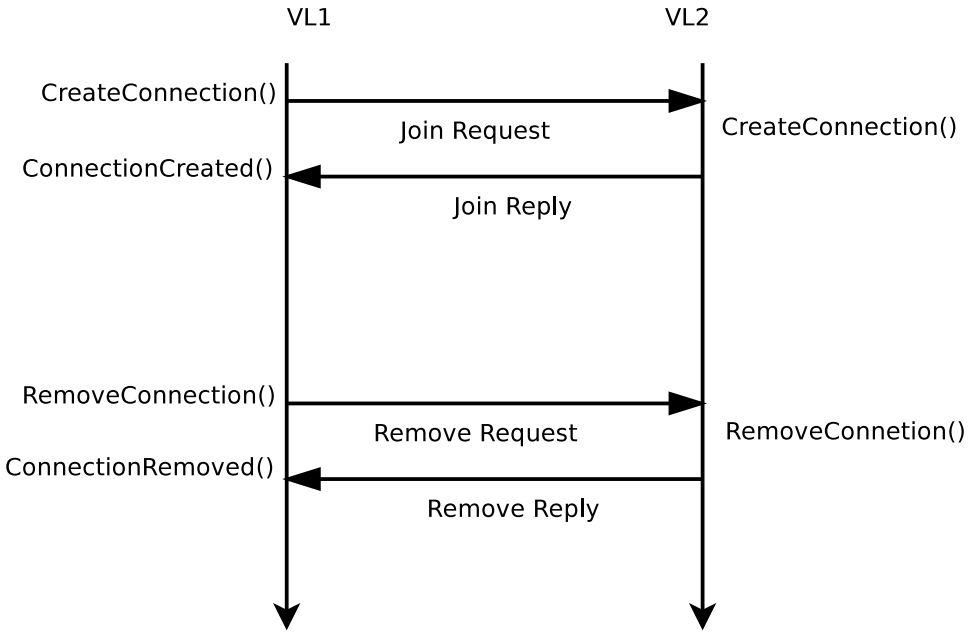
- Konfigurační element - popis konkrétního prvku v kolekci, včetně jeho vlastností.
- Kolekce elementů - popis konkrétního prvku v kolekci, včetně jeho vlastností.
- Konfigurační sekce - popis konkrétního prvku v kolekci, včetně jeho vlastností.

Příklad konfiguračního souboru aplikace lze nalézt v kapitole 5.

4.5 Komunikace

Pro komunikaci se vzdálenými instancemi Virtlabu, lokálními nadřazenými komponentami a uživatelem, slouží třídy *Comm* a *Control*. Instance těchto tříd jsou vytvořeny při volání metody *Setup* v hlavní třídě *Main*.

Koncept komunikace mezi instancemi Virtlabu je znázorněn na obrázku 18.



Obrázek 18: Komunikace mezi tunelovacími servery instancí Virtlabu

4.5.1 Třída Comm

Tato třída nabízí metody pro komunikaci se vzdálenými instancemi Virtlabu. Posílá zprávy vytvořené pomocí třídy *Msg*. Odesílání těchto zpráv je realizováno pomocí protokolu UDP, metodou asynchronous callback, kdy je pro odeslání vyhrazeno dynamicky vlákno, z poolu vláken.

Třída obsahuje tyto metody:

- *public Comm(IPEndPoint ep)* - Konstruktor třídy. Provadí vytvoření UDP socketu a následné svázání s ním. Zároveň se v něm spouští asynchroní naslouchání na tomto socketu.
  - IPEndPoint ep - koncový bod tunelovacího serveru lokální instance Virtlabu
- *public void Send(string data,MsgType type, int vlId, string connId, EndPoint ep)* - Vytváří a zahajuje odeslání zprávy vzdálené instanci Virtlabu.
  - string data - datová část zprávy, obsahuje string, který inicoval vytváření spojení
  - MsgType type - typ zprávy, více o třídě Msg v části věnované struktůrám
  - int vlId - identifikátor VLAN, která bude přenášet rámce spojení přes GRE tunel
  - string connId - identifikátor vytvářeného spoje
  - EndPoint ep - koncový bod vzdálené instance Virtlabu, na který bude zpráva odeslána
- *private void SendCB(IAsyncResult asr)* - Metoda volaná po dokončení odeslání zprávy.
- *private void ReceiveCB(IAsyncResult asr)* - Metoda volaná po přijetí zprávy. Vytváří z přijatých bytů instanci třídy *Msg*. Dále vytvoří instanci třídy *Task*, do které zabalí vytvořenou instanci zprávy, a zařazuje ji do fronty v třídě *Main*. Inicializuje také pokračování v běhu hlavního vlákna. Ihned poté pokračuje v naslouchání na zadaném socketu.

4.5.2 Třída Control

Tato třída nabízí metody pro komunikaci s komponentami lokální instance Virtlabu, případně uživatelem. Komunikace je realizována pomocí protokolu TCP. Použita je opět metoda asynchronous callback, kdy je pro odeslání vyhrazeno dynamicky vlákno, z poolu vláken.

Třída obsahuje tyto metody:

- *public Control (IPEndPoint ep)* - Konstruktor třídy. Provadí vytvoření TCP socketu, následné svázání s ním, a spuštění asynchroního naslouchání na tomto socketu.
  - IPEndPoint ep - koncový bod tunelovacího serveru lokální instance Virtlabu

- *private void AcceptCB (IAsyncResult asr)* - Přijme nové spojení a okamžitě spouští opětovné naslouchání na socketu.
- *private void ReceiveCB(IAsyncResult asr)* - Metoda volaná po přijetí zprávy. Vytváří z přijatého řetězce instanci třídy *Task*, do které zabalí přijatý řetězec, a zařazuje ji do fronty v třídě *Main*. Inicializuje také pokračování v běhu hlavního vlákna. Ihned poté pokračuje v naslouchání na zadaném socketu.
- *public void Send (Socket tcpclient, String data)* - Vytváří a zahajuje odeslání řetězce na TCP socketu připojeného klienta.
  - Socket tcpclient - TCP socket klienta
  - String data - řetězec určený k odeslání
- *private void SendCB (IAsyncResult asr)* - Volána po přijetí řetězce.
- *private string Parser(string input)* - Odstranuje z přijatého řetězce symboly, související s odesláním. Zbylý řetězec je vrácen k dalšímu zpracování.

Zde je nutno zmínit, že koncept komunikace mezi instancemi Virtlabu nebyla v původním konceptu implementována. K implemntaci toho konceptu vedla především snaha, o větší samostatnost aplikace tunelovacího serveru.

4.6 Zachytávání provozu

Zachytávání provozu nelze řešit přímo v Mono projektu, jelikož tento, z nižších vrstev ISO-OSI modelu, nativně podporuje pouze práci s 3. a 4. vrstvou. Tedy bylo nutno hledat řešení v podobě knihovny či externího nástroje.

Při hledání jsem se mimo jiné soustředil i na možné implementace knihovny *libpcap*, jež je k zachytávání provozu přímo určena. Nakonec se podařilo najít interface pro komunikaci nebo iplmentaci této knihovny pro jazyk C# - knihovnu *SharpPcap*. Tuto knihovnu vyvíjí Chris Morgan s přispěním mnoha nezávislých vývojářů. Je k dispozici jako Open Source a poskytuje přístup ke knihovně *libpcap* případně *winpcap*. Je tedy multiplatformním řešením vhodným jak pro původní .NET framework tak pro projekt Mono. Knihovna existuje již ve verzi 4.0.1 a je tedy dostatečně vyzrálá pro nasazení do provozu[9].

4.6.1 Třída PcapUtil

Samotná třída, sloužící k zachytávání provozu, má název *Pcaputil* a je umístěna v druhém projektu *TunnUtil*. Třída je statická.

Třída obsahuje tyto metody

- *private static int FindDevice(string deviceName)* - Najde zařízení a vrátí jeho index v kolekci možných zachytávacích zařízení.
  - string deviceName - jméno hledaného zařízení

- *public static void DumpToFile(string deviceName, string capFile)* - Metoda spustí zachytávání provozu na daném zařízení. Data budou ukládány do zadaného souboru. Metoda nejdříve najde rozhraní a přepne jej do promiskuitního modu. Následně vytvoří instanci třídy *CaptureFileWriterDevice*, která slouží k samotnému zápisu do souboru. Nakonec registruje událost *device_OnPacketArrival*, která bude veškerý příchozí provoz zapisovat pomocí vytvořeného zařízení do souboru.
  - string deviceName - jméno zařízení, na kterém se bude zachytávat provoz
  - string capFile - jméno souboru, do kterého budou zachycená data uložena
- *public static void StopDump(string deviceName)* - zastaví zachytávání na daném zařízení
  - string deviceName - jméno zařízení, na kterém bude ukončeno zachytávání provozu
- *private static void device_OnPacketArrival(object sender, CaptureEventArgs e)* - Událost při příchodu packetu na rozhraní, tento zapíše do souboru.

4.7 Databáze

Databáze v projektu zastupuje pouze určitou formu archivace navázaných spojení. Provozní data jsou ukládány ve strukturách přímo v paměti. Nicméně v budoucnu bude možno program rozšiřovat, tak lze použít tento, již vytvořený, modul.

V zájmu využití existující databáze typu MySQL, jsem hledal vhodné ORM s podporou pro tuto databázi, v ideálním případě založené na technologii Linq. Bohužel existující Open Source řešení nejsou ani zdaleka ve stavu vhodném pro nasazení (priklad). Existují však komerční řešení, jež jsou vyzrálá a plně funkční, avšak z podstaty nepoužitelná ve vytvářeném programu (priklad).

Pro demonstraci funkčnosti přístupu do MySQL databáze sem tedy zvolil přístup bez použití jakéhokoliv ORM. Pro účely této aplikace, je toto dostačující řešení.

4.7.1 Třída DatabaseUtil

Přístup k databázi řeší třída *DatabaseUtil*, která obsahuje tyto metody

- *public DatabaseUtil (string inConnString)* - Konstruktor třídy. Jako vstup má připojovací řetězec k databázi.
  - string inConnString - připojovací řetězec k databázi
- *private void DatabasePrepare()* - Vytváří tabulku, pro ukládání archivovaných spojení.
- *public void DatabaseInsert(Connection archivedConn)* - Po zavolání provede vložení archivovaného spojení do databáze.
  - Connection archivedConn - propoj určený k vložení do databáze

4.8 Volání externích nástrojů

Jednou z nejdůležitějších částí programu, je třída *ExternalToolsCalls*. Metody definované v této třídě zajištují volání nástrojů popsaných v kapitole 2. Samotné volání těchto nástrojů je řešeno vytvořením nového procesu, ve kterém se spustí daný nástroj. Projekt Mono převzal postup vytváření nových procesů z .NET frameworku, ze kterého je odvozen. V systému Windows je sice přítomen určitý druh uživatelský práv pro spouštění programů, avšak není zdaleka tak nedílnou součástí systému jako v případě operačního systému Linux. Proto není až tolik překvapivé, že podpora pro spouštění procesů s právy *roota* není v současné verzi projektu doimplementována. Tento nedostatek lze vyřešit bud spouštěním celé aplikace s patřičnými právy, nebo modifikovat práva přímo jednotlivých nástrojů, jež budou využívány.

4.8.1 Třída ExternalToolsCalls

Metody třídy *ExternalToolsCalls*

- *private string CreateVlanInt (string iFace, int vlanId)* - Vytvoří VLAN rozhraní s daným ID, na daném rozhraní. Vrácen je řetězec se jménem VLAN.
- *private void RemoveVlanInt (string iFace, int vlanId)* - Odstraní VLAN s daným ID z daného rozhraní.
- *private void CreateBrInt (string iFace1,string iFace2, int brId)* - Vytvoří rozhraní typu Bridge s daným ID. Následné k němu přiřadí dvě rozhraní určená k přemostění.
- *private void RemoveBrInt (int brId)* - Odstraní rozhraní typu Bridge s daným ID.
- *public string CreateGreTunnel (int greId, IPAddress local, IPAddress remote, int ttl)* - Vytvoří koncové rozhraní GRE tunel ze zadaných parametrů.
- *public void SetDev (string dev, bool flag)* - Nastaví dané rozhraní do zadaného stavu.
- *private void SetMtu (string dev, int mtu)* - Danému rozhraní nastaví zadanou hodnotu MTU.
- *public void MkRemoteLocalConnection (int brId, int locVlId, int tunnVlId, string locIface, string tunnIface)* - Vytvoří spojení mezi lokálním a vzdáleným zařízením. V rámci této metody jsou volány metody, zmíněné výše v popisu této třídy. V případě neúspěšného vytvoření některé komponenty spojení, jsou již vytvořené komponenty odstraněny.
- *public void MkRemoteRemoteConnection (int brId, int tunnVlId1, int tunnVlId2, string tunnIface1, string tunnIface2)* - Vytvoří spojení mezi dvěma vzdálenými zařízeními. V rámci této metody jsou volány metody, zmíněné výše v popisu této třídy. V případě neúspěšného vytvoření některé komponenty spojení, jsou již vytvořené komponenty odstraněny.

- *public void MkLocalLocalCapConnection (int brId, int locVlId1, int locVlId2, string locIface1, string locIface2)* - Vytvoří spojení mezi dvěma lokálními zařízeními s využitím zachytávání provozu. V rámci této metody jsou volány metody, zmíněné výše v popisu této třídy. V případě neúspěšného vytvoření některé komponenty spojení, jsou již vytvořené komponenty odstraněny.
- *public void RmRemoteLocalConnectionint (int brId, int locVlId, int tunnVlId, string locIface, string tunnIface, int progress)* - Odstraní spojení mezi lokálním a vzdáleným zařízením. V rámci této metody jsou volány metody, zmíněné výše v popisu této třídy.
- *public void RmRemoteRemoteConnection (int brId, int tunnVlId1, int tunnVlId2, string tunnIface1, string tunnIface2, int progress)* - Odstraní spojení mezi dvěma vzdálenými zařízeními. V rámci této metody jsou volány metody, zmíněné výše v popisu této třídy.
- *public void RmLocalLocalCapConnection (int brId, int locVlId1, int locVlId2, string locIface1, string locIface2, int progress)* - Odstraní spojení mezi dvěma lokálními zařízením s využitím zachytávání provozu. V rámci této metody jsou volány metody, zmíněné výše v popisu této třídy.
- *public void ClearAll ()* - Provede smazání všech rozhraní vytvořených aplikací.
- *private Process initProcess()* - Vytvoří nový proces, který poté využívají další metody třídy.

```
private void CreateBrInt (string iFace1, string iFace2, int brId)
    {

        StringBuilder sb = new StringBuilder ("addbr br");
        sb.Append (brId);

        Process workerProc = initProcess ("brctl", sb.ToString ());
        workerProc.Start ();

        workerProc.WaitForExit ();

        sb.Clear ();
        sb.Append ("addif br");
        sb.Append (brId);
        sb.Append (" ");
        sb.Append (iFace1);

        workerProc = initProcess ("brctl", sb.ToString ());
        workerProc.Start ();

        workerProc.WaitForExit ();


        sb.Clear ();
```

```
sb.Append ("addif br");
sb.Append (brId);
sb.Append (" ");
sb.Append (iFace2);

workerProc = initProcess (" brctl ", sb.ToString ());
workerProc.Start ();

workerProc.WaitForExit ();

}
```

Výpis 2: Metoda vytvářející rozhraní typu bridge

Ve výpisu kódu 2 je vidět metoda *CreateBrInt*. V této metodě je nejdříve vytvořen příkaz pro externí proces, který vytvoří rozhraní typu bridge pomocí nástroje *brctl*. Následně voláním metody *initProcess* vrácena instance reprezentující externí process - *workerProc*. Tento proces je následně spuštěn a vyčká se na jeho ukončení. Dále je vytvořen příkaz pro přídání rozhraní k nově vytvořenému rozhraní typu bridge. Instance procesu *worker-Proc* je opětovně spuštěna s jiným parametrem. Stejným způsobem je přidáno také druhé přemosťované rozhraní.

Obdobný příklad pro vytváření VLAN rozhraní je ukázán ve výpisu 6.

4.9 Logování

Pro logování je podpora integrována přímo v projektu Mono. Využívá k tomu přímého volání programu *Syslog*.

4.9.1 Třída LogUtil

V této třídě jsou obsaženy tyto metody:

- *public LogUtil (string id)* - Kostruktor třídy. Inicializuje logování do souboru definovaném zadaným identifikátorem a následně otevírá logování.
- *public void LogEvent(string msg, SyslogLevel level)* - Provede vytvoření záznamu se zadaným řetězcem a úrovní logu.
- *public void CloseLog()* - Uzavře logování.

```
public static void LogEvent (string msg, int level)
    {
        SyslogLevel syslevel;

        switch (level) {
        case 1:
            syslevel = SyslogLevel.LOG_CRIT;
            break;
        case 2:
```

```
            syslevel = SyslogLevel.LOG_INFO;
            break;
        case 3:
            syslevel = SyslogLevel.LOG_DEBUG;
            break;
        default:
            syslevel = SyslogLevel.LOG_CRIT;
            break;
        }

        Syscall.syslog (syslevel, msg);

    }
```

Výpis 3: Metoda zapisující události do systémového logu

Výpis zdrojového kodu 3 ukazuje metodu pro logování událostí. Fuknce metody spočívá ve zjištění, která úroven logu se má při zápisu zprávy použít a následně je volán zápis do *Syslogu*. Přímý zápis do *Syslogu* umožňuje metoda *Syscall*, které umožňuje některá přímá volání určených systémových volání systému Linux. Tato funkcionalita je implementována v namespace *Mono.Unix.Native*, který je vytvořen specialně pro projekt Mono a není obsažen v původním .NET frameworku.

4.10 SNMP

Pro práci s protokolem SNMP není v projektu Mono přímá nativní podpora. Lze ji však doimplementovat, díky možnosti práce s protokolem UDP.

Mnohem výhodnější je však použití některé z existujících Open Source knihoven, které poskytují API k práci s tímto protokolem. Pro implementaci byla zvolena knihovna *SharpSNMP*. Tuto knihovnu, nebo spíše sadu knihoven a nástrojů, vyvíjí tým nezávislých vývojářů. Mezi hlavní autory patří Malcolm Crowe a Lex Li. Tento multiplatformní projekt je vyvíjen již od roku 2008 a o jeho vyzrálosti, a vhodnosti pro nasazení, tedy není pochyb[8].

4.10.1 Třída SnmpUtil

V třídě *SnmpUtil* jsou obsaženy tyto metody

- *public void PortFlip(string port,bool state)* - Nastaví daný port do zadaného stavu.
- *public void PortChangeVlan(string port,int vlan)* - Nastaví VLAN zadaného portu na zadanou hodnotu.

V případě obou metod je nejdříve zjištěno ID portu. Nejdříve je vrácen seznam portů pomocí *SnmpWalk*. Výsledek je porovnán se zadaným jménem portu a je získáno jeho ID. Následně je nastaven stav portu nebo nastavena hodnota *VLAN* portu.

4.11 Struktury

Při implemntaci aplikace bylo nutno vytvořit mnoho doplňkových struktur pro zpřehlednění a zjednodušení kodu i zjednodušení samotné implementace. Jedná se především o strukturu k ukládání dat, načtených z konfiguračního souboru, dále pak struktury držící data o vytvořených propojích a v neposlední řadě také struktury zpráv a úloh zasílaných mezi vlákny aplikace.

Mezi tyto struktury patří:

- *public class Task* - Instance třídy je vytvořena třídami pro komunikaci a uložena do fronty, kde ji vyzvedne hlavní třída *Main* resp. hlavní vlákno aplikace. Ve třídě jsou ve třídě jsou uloženy příchozí data, určena ke zpracování.
- *public class Msg* - Třída reprezentuje zprávu, zasílanou mezi instancemi jednotlivými tunelovacími servery instancí Virtlabu. Metody třídy slouží k vytvoření struktury zprávy z přijatých bytů a ke zpětnému převodu zprávy na pole bytů.
- *public class Device* - Zařízení, přípojené k multiplexeru lokální instance Virtlabu.
- *public class Connection* - Informace o jednom konkrétním propoji. Instance třídy jsou uloženy v tabulce propojů.
- *public class Bridge* - Reprezentace rozhraní typu most.
- *public class Vlan* - Rozhraní typu VLAN.
- *public class TrunkIface* - Třída reprezentuje rozhraní k Trunk lince Multiplexeru. Součástí třídy je pole instancí třídy *Vlan*, které zaznamenává VLAN zachytávané na tomto rozhraní.
- *public class TunnIface* - Třída reprezentuje rozhraní typu GRE tunel, propojující jednotlivé instance Virtlabu skrze internet. Součástí třídy je opět pole instancí třídy *Vlan*, které zaznamenává VLAN zachytávané na tomto rozhraní.
- *public class Mux* - Multiplexer, připojený Trunk linkou k lokální instanci Virtlabu. Třída obsahuje pole instancí třídy Device, které určují zařízení připojená ke konkrétnímu Multiplexeru. Rovněž obsahuje instanci třídy *SnmpUtil*, sloužící k správě tohoto Multiplexeru skrze protokol SNMP.
- *public class VirtlabInstance* - Reprezentuje vzdálenou instanci Virtlabu.

Ve všech těchto strukturách resp. třídách, jsou také vytvořeny kontruktory a běžné sady metod typu *Get* a *Set*.

```
public Msg (byte[] inputData)
        {
            this.type = (MsgType)BitConverter.ToInt32 (inputData, 0);
            int lenghtOfData = BitConverter.ToInt32 (inputData, 4);
            int lenghtOfConnId = BitConverter.ToInt32 (inputData, 8);
```

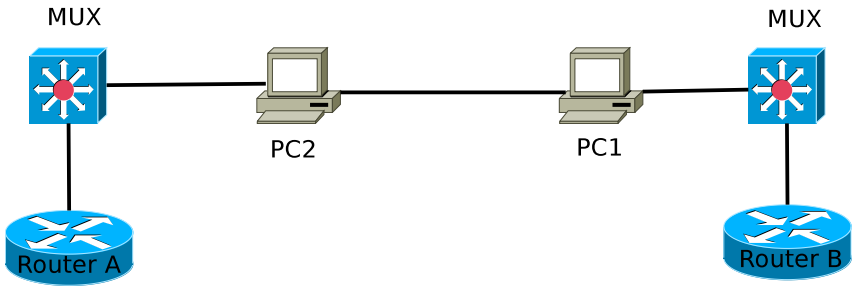
```
        int lenghtOfTimeStamp = BitConverter.ToInt32 (inputData, 12);
        int lenghtOfEp = BitConverter.ToInt32 (inputData, 16);
        this.vIId = BitConverter.ToInt32 (inputData, 20);
        this.connId = Encoding.UTF8.GetString (inputData, 24, lenghtOfConnId);
        this.data = Encoding.UTF8.GetString (inputData, 24 + lenghtOfConnId, lenghtOfData);
        this.timeStamp = DateTime.Parse (Encoding.UTF8.GetString (inputData, 24 +
            lenghtOfData + lenghtOfConnId, lenghtOfTimeStamp));
        string[] temp = ParseEp (Encoding.UTF8.GetString (inputData, 24 + lenghtOfData +
            lenghtOfConnId + lenghtOfTimeStamp, lenghtOfEp));
        this.ep = new IPEndPoint (IPAddress.Parse (temp [0]), Int32.Parse (temp [1]));
    }
```

Výpis 4: Přetížený konstruktor třídy Msg

V ukázkovém výpisu zdrojového kódu 4 lze vidět vytvoření instance třídy *Msg* ze zadaného pole bytů. Nejdříve jsou vytvořeny položky pevné délky, následně pak položky s proměnnou délku, jejichž délka je určena v již vytvořených položkách.

5 Testování

Pro testování byla vytvořena následující topologie:



Obrázek 19: Testovací topologie

Použitá PC v rolích tunelovacích serverů měla nainstalován operační systém Ubuntu ve verzi 11.10. V roli Multiplexerů byly použity Multilayer přepínače Cisco 3560.

Po nainstalování prostředí Mono verze 2.10.8 a MySQL serveru verze 5.1 z repozitářů bylo nutné nastavit samotnou aplikaci a Multiplexery.

Použití protokolu SNMPv3 lze na přepínači nastavit pomocí těchto příkazů:

mux(config)#snmp-server group GROUP1 v3 priv

mux(config)#snmp-server user User1 GROUP1 v3 auth md5 pass456 priv des pass456

Na multiplexeru jsou dále nastaveny porty do patřičných režimů, těmto portům přiřazeny VLAN a zapnuto tunelování protokolu CDP.

Na samotném stroji tunelovacího serveru je nutné upravit spouštění nástrojů *iproute2*, *modprobe* a *brctl*. V případě, že některý z nich není nainstalován, je nutno toto napravit.

Dále je vhodné upravit link ke knihovně *libpcap* v mapovacím souboru knihovny *SharpPcap*.

Nutné je také nastavení *Syslogu* například podle instrukcí na oficiálních stránkách projektu Virtlab[5].

Ukázka vzorového konfiguračního souboru aplikace:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <configSections>
    <section name="vllistsection" type="CoreLogic.CustomConfigurationSectionLoc, CoreLogic"
      requirePermission="false" />
    <section name="devicelistsection" type="CoreLogic.CustomConfigurationSectionDev,
      CoreLogic" requirePermission="false" />
  <section name="muxlistsection" type="CoreLogic.CustomConfigurationSectionMux,
    CoreLogic" requirePermission="false" />
  </configSections>
  < vllistsection >
    <vls>
      <add name="Opava" ip="10.0.0.2" port="30000" />
    </vls>
  </ vllistsection >
  <devicelistsection>
```

```
    <devices>
      <add id="R1_fe0/0" mux="MUX1" vlan="10" iface="FastEthernet0/1" />
    </devices>
  </devicelistsection>
  <muxlistsection>
    <muxes>
      <add id="MUX1" iface="eth1" ip="10.0.0.1" snmpsett="public;1000;0;MD5;pass456;DES;
          pass456;User1" />
   </muxes>
  </muxlistsection>
  <connectionStrings>
      <add name="connectionString1" connectionString="Data Source=mix1;Initial Catalog=DB;
          Integrated Security=True" providerName="System.Data.SqlClient" />
  </connectionStrings>
  <appSettings>
   <add key="vlLocalName" value="Ostrava" />
    <add key="localIp" value="10.0.0.1" />
   <add key="greTtl" value="255" />
    <add key="tcpCommandPort" value="40001" />
    <add key="udpComunicationPort" value="30001" />
    <add key="udpRepeatCount" value="5" />
   <add key="snmpManIf" value="eth1" />
   <add key="defaultDebugLevel" value="0" />
  </appSettings>
</configuration>
```

Výpis 5: Příklad konfiguračního souboru

Spuštění samotné aplikace lze provést buďto jako klasickou konzolovou aplikaci pomocí příkazu : *mono CoreLogic.exe* Alternativně jako *mono-service*: *mono-service CoreLogic.exe*

Následně lze aplikaci ovládat například pomocí programu *telnet* na nakonfigurované adrese a portu.

Ukázka z provozu, zachyceného programem *Wireshark*:

```
▸ Ethernet II, Src: Cisco_6e:c4:18 (00:23:eb:6e:c4:18), Dst: PlanetTe_3b:6b:63 (00:30:4f:3b:6b:63)
▸ Internet Protocol, Src: 172.16.1.2 (172.16.1.2), Dst: 192.168.1.2 (192.168.1.2)
▸ Generic Routing Encapsulation (Transparent Ethernet bridging)
▸ Ethernet II, Src: Cisco_b4:a1:4a (00:1e:be:b4:a1:4a), Dst: Cisco_b4:ad:e8 (00:1e:be:b4:ad:e8)
▸ 802.1Q Virtual LAN, PRI: 0, CFI: 0, ID: 20
▸ Internet Protocol, Src: 10.0.0.2 (10.0.0.2), Dst: 10.0.0.1 (10.0.0.1)
▸ Internet Control Message Protocol
```

Obrázek 20: Zachycený ICMP packet

Pří testování bylo zjištěno :

- aplikace je stabilní při běžném provozu
- výkon je dostatečný pro rozumný počet spojení - toto je zásvislé na implementaci použitých modulů jádra systému Linux a na použitém hardwaru

- počet nedoručených packetů je mnohem nižší než v případě původního řešení - toto opět závisí především na použitém hardwaru, nicméně lze konstatovat, že vytvořená aplikace je v tomto ohledu vždy výkonější
- rychlost vytváření propojů je nižší - toto je očekávaná vlastnost, vzhledem k větší složitosti procesu vytváření propojů
- stabilita propojů není závislá na aplikaci - v případě pádu aplikace zůstávají propoje vytvořeny až do nového spuštění aplikace

6 Nasazení a budoucí vývoj

Nasazení komponenty do reálného provozu prokázalo výhody, vytyčené na začátku práce. Vzrostla především stabilita propojů a účinost tunelování rámců. Problémem je mnohdy komplikovanější nasazení, z důvodu nutnosti doinstalování běhového prostředí projektu Mono na jednotlivé instance Virtlabu resp. jejich tunelovací servery. Také počáteční nastavení konfigurovacího souboru je složitější než u původního řešení.

Nicméně komponenta funguje korektně a je možno do budoucna implementovat verzi Virtlabu, založenou na projektu Mono, resp. jazyku C#, se základem v této komponentě. Jako optimální by se v tomto směru jevilo nahrazení použitých externích nástrojů kódem využívajícím NETLINK socket, implementovaném buď v jazyku C#, případně C++. Nutnou prerekvizitou je dobré zdokumentování protokolu NETLINK a modulů, potřebných k vytváření propojů.

7 Závěr

Díky této práci jsem se dozvěděl mnohé z fungování síťové části operačního systému Linux. Zjistil jsem nedostatky v dokumentaci, které však nelze jednoduše napravit a bylo by potřeba vypsání samostatné práce na toto téma.

Dále jsem analyzoval současný stav komponenty tunelovacíhi serveru a navrhnul řešení nové. Při hledání prostředků pro toto řešení jsem zběžně rozebral nejznámější technologie tunelování provozu na 2. vrstvě ISO-OSI modelu v prostředí operačního systému Linux.

Také jsem prozkoumal možnosti a vyzrálost projektu Mono, především tedy jeho částí potřebných pro implementaci jež byla cílem této práce. Výsledkem bylo zjištění, že projekt Mono je sice relativně vyzrálý a vhodný k nasazení, nicméně podpora funkcí psecifických pro systém Linux, není ještě kompletní.Vyhledem k tomu, že jde o Open Source projekt, který nemá, v případě současného tempa vývoje, šanci držet krok s .NET frameworkem, je tento fakt vcelku pochopitelný a tolerovatelný. Projekt však díky kompatibilitě s .NET frameworkem, získává přístup k množství multiplatformně použitelných Open Source knihoven. Dvě z těchto knihoven jsem také využil při realizaci práce.

Celkově lze tedy konstatovat, že řešení splnilo požadavaky na něj kladené. Zároveň může sloužit k popularizaci rozvíjejího se projektu Mono.

Hlavním přínosem pro mě byla možnost pracovat na projektu Virtlab a zároveň si vyzkoušet, pro mě netradiční a neznámý, vývoj aplikací pomocí jazyku C# na platformě operačního systému Linux.

Daniel Stříbný

8 Literatura

[1] BENVENUTI, Christian. Understanding Linux network internals. USA : O Reilly Media, Inc., 2006. 1035 s. ISBN 9780596002558.

[2] BECK, Michael. Linux kernel internals. 2nd. USA : Addison-Wesley, 1998. 480 s. ISBN 9780201331431.

[3] MAMONE, Mark. Practical Mono. USA : Apress, 2005. 424 s. ISBN 9781590595480.

[4] GUNNERSON, Eric. A Programmer's Introduction to C# 2.0, Third Edition. USA : Apress, 2005. 568 s. ISBN 9781590595015.

[5] Vývojový tým projektu Virtlab. Oficiální stránka projektu Virtlab [online]. [cit. 2012-04-04]. Dostupné na World Wide Web: http://www.virtlab.cz/.

[6] XAMARIN. Oficiální stránka projektu Mono [online]. [cit. 2012-04-04]. Dostupné na World Wide Web: http://www.mono-project.com/.

[7] IETF. IETF Tools [online]. 2012-02-21 [cit. 2012-04-04]. Dostupné na World Wide Web: http://tools.ietf.org/.

[8] CROWE. C# Based Open Source SNMP for .NET and Mono [online]. 2012-03-21 [cit. 2012-04-04]. Dostupné na World Wide Web: http://sharpsnmplib.codeplex.com/.

[9] MORGAN. SharpPcap [online]. 2012-04-13 [cit. 2012-04-20]. Dostupné na World Wide Web: http://sourceforge.net/projects/sharppcap/.

[10] LINUX FOUNDATION TEAM. Linux Foundation [online]. [cit. 2012-04-20]. Dostupné na World Wide Web: ttp://www.linuxfoundation.org/.

A Obsah CD

Následující tabulka popisuje umístění souborů na CD a jejich popis.

Adresář	Popis
Dokument	Diplomová práce ve formátu souboru pdf
DokumentLaTeX	Zdrojové kódy diplomové práce ve formátu souboru tex
TunelovaciServer	Spustitelný soubor, který reprezentuje výsledný program
UserGuied	Návod k nastavení a použití
Src	Zdrojové kódy diplomové práce v jazyku C#

Tabulka 1: Obsah CD

B Zdrojové kódy

```
private string CreateVlanInt (string iFace, int vlanId)
    {

        string output = null;

        StringBuilder sb = new StringBuilder (" link add link ");
        sb.Append (iFace);
        sb.Append (" name ");
        sb.Append (iFace);
        sb.Append (".");
        sb.Append (vlanId);
        sb.Append (" type vlan id ");
        sb.Append (vlanId);

        Process workerProc = initProcess ("ip", sb.ToString ());
        workerProc.Start ();

        workerProc.WaitForExit ();
        if (workerProc.ExitCode != 1) {
            throw new Exception ("Error in external process");

        }

        sb.Clear ();
        sb.Append (iFace);
        sb.Append (".");
        sb.Append (vlanId);

        output = sb.ToString ();

        return output;
    }
```

Výpis 6: Vytvoření VLAN rozhraní

```
private static void CreateConnection (string[] input, Socket socket)
    {

        Connection newconnection = new Connection ();
        newconnection.Id = GenerateID ();
        newconnection.VlanId1 = System.Int32.Parse (input [0]);
        newconnection.VlanId2 = System.Int32.Parse (input [1]);
        newconnection.CreationCommand = string.Join (" ", input);

        if (input [2] == localIP.ToString () && input [3] != null) {
            if (!(connections.Exists (x => x.VlanId1 == newconnection.VlanId1 && x.
                ConnectionType == ConnectionType.Local_Only)))
                string tmp_iface1 = muxes.Find (x => x.Name == (localDevices.Find (y => y.
                    DeviceVlanId == newconnection.VlanId1).DeviceMux)).Iface;
                string tmp_iface2 = muxes.Find (x => x.Name == (localDevices.Find (y => y.
                    DeviceVlanId == newconnection.VlanId2).DeviceMux)).Iface;
```

```
                if (trunkIfaces.Find (x => x.Name == tmp_iface1).TrunkVlans.FindIndex (y => y
                    .Id == newconnection.VlanId1) != -1) {
                    trunkIfaces.Find (x => x.Name == tmp_iface1).TrunkVlans.Find (y => y.Id ==
                        newconnection.VlanId1).VlanStatus = Status.Active;
                } else {
                    trunkIfaces.Find (x => x.Name == tmp_iface1).TrunkVlans.Add (new Vlan (
                        newconnection.VlanId1, Status.Active));
                }

                if (trunkIfaces.Find (x => x.Name == tmp_iface2).TrunkVlans.FindIndex (y => y
                    .Id == newconnection.VlanId2) != -1) {
                    trunkIfaces.Find (x => x.Name == tmp_iface2).TrunkVlans.Find (y => y.Id ==
                        newconnection.VlanId2).VlanStatus = Status.Active;
                } else {
                    trunkIfaces.Find (x => x.Name == tmp_iface2).TrunkVlans.Add (new Vlan (
                        newconnection.VlanId2, Status.Active));
                }
                Bridge b;
                int tmp_br_id;
                int freeBridgeIndex = bridges.FindIndex (x => x.Status == Status.Inactive);
                if (freeBridgeIndex == -1) {
                    b = new Bridge (bridges.Count + 1, tmp_iface1, tmp_iface2, Status.Active);
                    bridges.Add (b);
                    tmp_br_id = b.Id;
                } else {
                    bridges [freeBridgeIndex].Status = Status.Active;
                    bridges [freeBridgeIndex].IFace1 = tmp_iface1;
                    bridges [freeBridgeIndex].IFace2 = tmp_iface2;
                    tmp_br_id = bridges [freeBridgeIndex].Id;
                }
                etc.MkLocalLocalCapConnection (tmp_br_id, newconnection.VlanId1,
                    newconnection.VlanId2, tmp_iface1, tmp_iface2);
                TunnUtil.PcapUtil.DumpToFile ("br" + tmp_br_id, newconnection.Id);
                newconnection.Bridge = tmp_br_id;
                newconnection.ConnectionStatus = ConnectionStatus.Active;
                newconnection.Capture = true;
                newconnection.ConnectionType = ConnectionType.Local_Only;
                newconnection.Created = DateTime.Now;
                connections.Add (newconnection);

                tcpControl1.Send (socket, "OK \n");
            }
            }

        }

    }
```

Výpis 7: Část metody pro inicializaci procesu vytváření propoje